A Padawan Programmer's Guide to Developing Software Libraries

Yurkovich, James T.; Yurkovich, Benjamin J.; Dräger, Andreas; Palsson, Bernhard O.; King, Zachary A.

Published in:
Cell Systems

Link to article, DOI:
10.1016/j.cels.2017.08.003

Publication date:
2017

Document Version
Peer reviewed version

Please cite this article in press as: Yurkovich et al., A Padawan Programmer's Guide to Developing Software Libraries, Cell Systems (2017), https://doi.org/10.1016/j.cels.2017.08.003

Cell Systems
Commentary

A Padawan Programmer's Guide to Developing Software Libraries

James T. Yurkovich,[1,2] Benjamin J. Yurkovich,[3,4] Andreas Dräger,[5] Bernhard O. Palsson,[1,2,6,7] and Zachary A. King[1,*]
[1]Bioengineering Department, University of California, San Diego, La Jolla, CA 92093, USA
[2]Bioinformatics and Systems Biology Program, University of California, San Diego, La Jolla, CA 92093, USA
[3]Technicity, Columbus, OH 43212, USA
[4]Center for Automotive Research, The Ohio State University, Columbus, OH 43212, USA
[5]Applied Bioinformatics Group, Center for Bioinformatics Tübingen (ZBIT), University of Tübingen, 72076 Tübingen, Germany
[6]Department of Pediatrics, University of California, San Diego, La Jolla, CA 92093, USA
[7]Novo Nordisk Foundation Center for Biosustainability, Technical University of Denmark, Kemitorvet, Building 220, 2800 Kongens, Lyngby, Denmark
*Correspondence: zaking@ucsd.edu
https://doi.org/10.1016/j.cels.2017.08.003

With the rapid adoption of computational tools in the life sciences, scientists are taking on the challenge of developing their own software libraries and releasing them for public use. This trend is being accelerated by popular technologies and platforms, such as GitHub, Jupyter, R/Shiny, that make it easier to develop scientific software and by open-source licenses that make it easier to release software. But how do you build a software library that people will use? And what characteristics do the best libraries have that make them enduringly popular? Here, we provide a reference guide, based on our own experiences, for developing software libraries along with real-world examples to help provide context for scientists who are learning about these concepts for the first time. While we can only scratch the surface of these topics, we hope that this article will act as a guide for scientists who want to write great software that is built to last.

There is a growing need for software that interacts with and utilizes biological data, whether through computation, visualization, or data storage. Researchers in the life sciences have been developing their own software tools to meet these needs, often with great success. Developing and maintaining software with many outside users is difficult work. Extensive learning resources for computer programming and software development make it easy to get started, but it remains challenging to develop high-quality software that is widely used. To gain traction, software libraries need to be intuitive, well documented, compatible, and extensible.

Several excellent articles have addressed a wide range of topics for software developers and bioinformaticians, with discussions ranging from best practices for scientific computing more generally (Wilson et al., 2014) to guidelines for scaling up bioinformatics analyses (Dudley and Butte, 2009) and suggestions for training in scientific computing and bioinformatics (Leprevost et al., 2014; Via et al., 2013). Several shorter articles have specifically addressed important ideas of which to be cognizant when developing software tools (Altschul et al., 2013; List et al., 2017; Loman and Watson, 2013; Prlić and Procter, 2012).

In this article, we provide a starting point and reference guide for scientists with some background in coding who are interested in developing and sharing their own software libraries. We present this guide through a series of lessons that have been influenced by our own experiences in developing software, covering topics from project scope and design to licensing and marketing (Figure 1). We hope that this article will be a useful roadmap not just at the beginning of the development process but also throughout the actual prototyping of code and dissemination of a software library. We can only hope to scratch the surface of these topics, but we provide references throughout for those interested in learning more about a particular topic. While most of the software that we highlight in this article is academic, the lessons that we discuss are applicable to developing both academic and commercial software. Finally, it should be noted that there is no correct way to go about developing software; this article simply outlines one tried-and-tested workflow.

The article is structured according to the workflow laid out in Figure 1. Lessons 1 and 2 cover the things you need to think about before starting development. Lessons 3–11 focus on the actual software development process, including planning, design, and implementation. Finally, Lessons 12–14 discuss important issues related to code distribution.

Lesson 1: Identify a Need for a New Piece of Software

The first step is to determine whether the software you intend to develop already exists—why invest time reinventing the wheel? Writing a software library from scratch is not necessarily (if ever!) a trivial endeavor, and you will probably save time if you can contribute to an existing project. Do a thorough search to educate yourself on what is already out there, the pros and cons of existing tools, and how new software could fill an unoccupied niche. If you run across a piece of software that might be relevant, make sure to read the documentation; the full range of capabilities for a piece of software may not be immediately obvious. It is important to remember that adapting or contributing to existing software can also greatly benefit the field by coalescing features and contributions into a centralized

Please cite this article in press as: Yurkovich et al., A Padawan Programmer's Guide to Developing Software Libraries, Cell Systems (2017), https://doi.org/10.1016/j.cels.2017.08.003

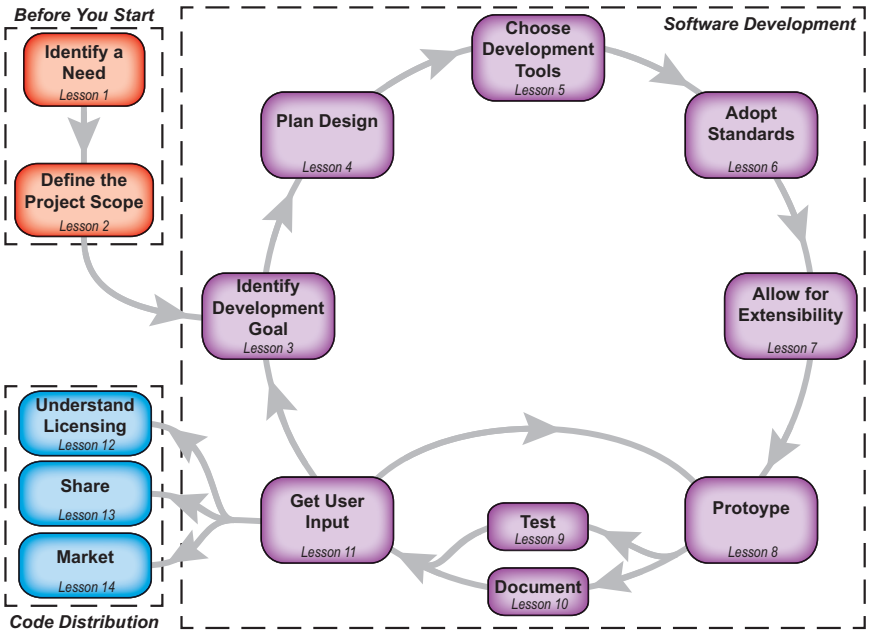

Figure 1. A Visual Description of How the 14 Lessons Discussed in This Article Fit Together during the Software Development Process

project that may already have an active community of users.

That is not to say that you should shy away from developing your own version of an existing tool. We began developing a tool for visualizing metabolic pathways (King et al., 2015) even though dozens of similar tools already existed in the form of both desktop applications (Droste et al., 2013; Funahashi et al., 2008; Smoot et al., 2011) and web applications (Chung et al., 2005; Kelder et al., 2012; Kono et al., 2009). Ultimately, we decided that existing software did not satisfy our particular needs, so we made a gamble that spending time to develop a new software library would be a worthwhile endeavor. Our tool, Escher, has since become a popular choice for pathway visualization. Deciding whether to develop new software is a judgment call, and you must weigh the risks and potential rewards. If you have trouble deciding, build a quick prototype and send it to potential users for feedback.

Lesson 2: Define the Scope of Your Project

At this point, you have already performed a thorough search and decided to officially embark on the development of your own software library. Before you start, though, it is important to determine the scope of the project. The scope should lay out the challenges your software will address and, just as importantly, the challenges your software will *not* address. As the project develops and new ideas are generated, this outline will enable you to stay on task and move forward within the confines of the previously identified project boundaries. If someone proposes an esoteric feature—which can happen frequently—you can easily explain that the requested feature lies outside the project scope.

If you plan to develop your software collaboratively, this is also the time to think about an appropriate infrastructure for good team management. As you determine your project's scope, start to think about how best to organize your team. Instant messenger services like Slack (https://slack.com), shared calendars for team meetings, and project-management platforms for software development such as Trello (https://trello.com) or PivotalTracker (https://www.pivotaltracker.com) will help keep your team on the same page throughout the rest of the development process.

Lesson 3: Identify Achievable Development Goals

After you have determined the scope of the overall project, the next step is to identify an achievable development goal. Your first development goal is to build the set of features for a minimum viable product (MVP). The MVP is the development point at which your software has just enough features to get feedback and traction from potential users. Think of this as a small target within the larger scope your project. The first milestones during the development process should be directed toward creating the MVP. This goal helps focus development toward achieving a working prototype, keeps the project from becoming too ambitious early on, and allows for realistic goals.

Defining the MVP can be challenging, and it requires you to see the bigger picture. Remember: the goal is to define the minimum state at which your software is *viable*—not the point at which it has a bunch of bells and whistles. If you are designing a starship for cargo transport, your MVP should not be a ship that can complete the Kessel Run in under 12 parsecs (unless you need to avoid Imperial forces). Rather, you should focus on meeting the minimum design specification to transport goods. There is no exact formula for defining the MVP, but talking to potential users is always a good first step. After you have reached your MVP (i.e., you have already been around the loop in Figure 1 at least once), identify the next development goal and focus your efforts there. At this point, you might create a list of several desired features, prioritize them, and target them one by one.

Lesson 4: Plan Your Design Based on Accepted Patterns

An important aspect of popular, open-source software libraries is the ability for users to comprehend the structure and execution of the underlying code. One of the best ways to ensure code accessibility is to use "design patterns." Inspired by reusable design patterns in architecture, software engineers have developed collections of well-defined and standardized components (Hunt and Thomas, 1999; Johnson et al., 1995). One example is the well-known model-view-controller pattern: it suggests separating user interfaces (views) from data structures (models) and application logic (controllers). Thinking about different conceptual design patterns before you start coding will help define how users interact with

Please cite this article in press as: Yurkovich et al., A Padawan Programmer's Guide to Developing Software Libraries, Cell Systems (2017), https://doi.org/10.1016/j.cels.2017.08.003

your software. Using design patterns helps to combat the potential existence of so-called anti-patterns (i.e., common trends that develop within code that hinder productivity) that can develop in your code (Brown, 1998).

At first glance, structuring software based on such design patterns might seem to require additional effort, but doing so can be advantageous. When handing your code over to other developers, the naming conventions of various patterns will be informative enough for others to intuitively understand what the code will do. Furthermore, you do not need to spend a long time developing your own concepts about how program components should fit together. Each language or framework might also have best practices for software architecture, so look for tips in the documentation.

Lesson 5: Choose Your Development Tools

You have a development goal. You have a design plan. Now it is time to start thinking about how you will implement your code. New software tools and languages are constantly being developed, making it difficult for developers to keep pace, a challenge referred to as "tool fatigue." Tool fatigue represents a constant hurdle, making a choice such as your primary development environment platform an important one. While it is fun to use the newest and flashiest tools out there, keep in mind that new tools might not continue to be supported over time, or they might have serious bugs that are not clear at first glance. Well-established, stable software is often a safe choice, but—especially for fast-moving platforms like the web—it might become outdated. When faced with this choice, you might want to seek advice from software developers who work in the area.

There are many of aspects of choosing a development language and platform to consider, so let us consider a practical example of how tool fatigue can influence the development process. Many researchers in the area of computational metabolic modeling use constraint-based reconstruction and analysis (COBRA) methods. Our group developed the first COBRA software library in MATLAB (Schellenberger et al., 2011), and it remains the most popular choice for the community. However, MATLAB presented several fundamental limitations (including the fact that it is costly commercial software), prompting the development of a new COBRA library in Python. This new library, COBRApy (Ebrahim et al., 2013), takes advantage of the community using Python for scientific computing and data science while also opening the door for users who may not have access to MATLAB. With the success of COBRApy, there is now interest in developing new COBRA libraries for users who prefer other languages, such as Julia (Heirendt et al., 2017).

While the move from MATLAB to Python enabled a host of new COBRA applications, the existence of two—and possibly three—versions of the same basic software library means that work has been duplicated. Having a software library available in multiple languages expands the potential user base, but communities that use one toolbox will likely have access to only a subset of all available features (it is unlikely for academic groups to be able to provide identical support for multiple platforms). If the latest methods are developed in a specific language (Bordbar et al., 2017, 2014), then users may be forced to learn a new language in order to use a particular feature. Thus, there is a delicate balance between choosing to adopt new tools and sticking with an old favorite.

Lesson 6: Adopt or Develop Standards

With the breadth of data types and corresponding analytical workflows in the life sciences, it is likely that your software library will need to interact with an existing software library or database. One of the biggest hurdles for those attempting to link different pieces of software is standardization—both in coding practice (Brazma et al., 2006; Stanford et al., 2015) and in how data is annotated and stored (Dräger and Palsson, 2014). Adopt the use of an existing standard if at all possible, as this will enable the use of common tools and make writing documentation much easier. If the software you are developing necessitates the development of new standards, think carefully about the potential advantages and disadvantages of the convention you are defining.

Standards also help combat potential compatibility issues that arise from versioning. When you update your software in a new release, remember that your changes might break other people's code. It is therefore important to communicate to your users how and when you make changes that might affect other software dependencies. The most convenient way to articulate such changes is through the version number of each release. A widely adopted approach called "semantic versioning" (http://semver.org) outlines specific version number requirements. Semantic versioning is a numerical classification for a software that has three components, such as version 2.1.0. The first number (major) signifies incompatible changes to interfaces and standards. The second number (minor) represents updates that are backward-compatible, while the third number (patch) designate backward-compatible bug fixes. Thus, versions 2.1.1 and 2.3.0 are expected to work the same as version 2.1.0, but version 3.0.0 will include changes that will break previous installations. Using semantic versioning therefore communicates all this information to a user through just the version number of a specific release.

Lastly, it is also important to consider internal standards, especially the use of version control. Version control systems—such as git, mercurial, and subversion—are such a powerful form of standardization that they are used for almost every major software project, public or private, large or small. Web applications like GitHub (https://github.com) and BitBucket (https://bitbucket.org/) add additional features like reporting bugs, questions, and contributions; they are used widely for academic software (Lesson 13). Version control systems have a significant learning curve, but many excellent resources exist for getting started (e.g., http://guides.github.com/activities/hello-world), and the benefits of adopting version control are difficult to overstate.

Lesson 7: Make Your Software Extensible

Another way to help ensure the longevity of your software is to design it with extensibility in mind. An "extensible" codebase is one that can easily be adapted to support unforeseen use cases. This is a critical aspect of writing code that can be used and developed by others, because

Please cite this article in press as: Yurkovich et al., A Padawan Programmer's Guide to Developing Software Libraries, Cell Systems (2017), https://doi.org/10.1016/j.cels.2017.08.003

Table 1. Tutorials Illustrating Some of the Principles Discussed in This Article

Tutorial for Accessing Web Services	A starter tutorial for bioscientists and engineers wanting to access data sources in an automated fashion using web services.
https://github.com/bjyurkovich/rest-tutorial-for-bio-science	
Tutorial and Example for Accessing and Extending Existing Web Interfaces	A demonstration project highlighting the application of the "Tutorial for Accessing Web Services" with discussion on how it was approached and implemented.
https://github.com/bjyurkovich/escher-interactive	
Tutorial for Building Extensions for Escher	Introduction to extending Escher by adding custom text, images, and plots to tooltips on the Escher pathway visualization.
http://escher.readthedocs.io/en/latest/development-tutorial.html	

it will help contribute to the utility and longevity of your tool. Some of the most popular academic software are excellent models of extensibility, from the Cytoscape App Store (Shannon et al., 2003) to applications that follow Unix guidelines for command line extensibility (Dudley and Butte, 2009) to the extensive use of web services by NCBI (Altschul et al., 1997; Federhen, 2012; Geer et al., 2010) and the Protein Data Bank (PDB) (Berman et al., 2000).

One easy way to build extensible code is to utilize design patterns (Lesson 4) and standards (Lesson 6) whenever possible. If your software is compatible with common tools and design patterns, it will be easy for someone familiar with standard tools to quickly understand what you did and how you did it. During the development of Escher, we wanted to make extensibility a priority, so we focused on developing standard interfaces. We made sure that users familiar with JavaScript could easily launch an Escher map within their own website, and we were careful to expose all Escher options to users. We also wanted to give users the chance to extend the visualization itself, so we created a standard interface for embedding custom tooltips that display extra information when users hover over elements within the visualization. You can see the effect of both decisions by comparing the standard Escher website to ErythroDB, a new knowledge base for the human red blood cell that uses the standards developed within Escher extensively (http://erythrodb.org). These extensibility features also made it possible for the Protein Data Bank (PDB) to include Escher in their website for connecting pathways to protein structures (www.rcsb.org) (Berman et al., 2000).

Extensibility comes in many forms, and each project requires consideration of the best way to enable extensions while still completing the project with available resources. As this topic can be difficult to grasp without concrete examples, we demonstrate what we mean by "extensibility" through a series of tutorials using Escher (Table 1). These examples show basic extensions that enable users to get started creating their own custom extensions and provide details for a few Escher extensions that have already been created, such as Escher-Interactive (http://escher-interactive.ucsd.edu).

Lesson 8: Develop Code while Maintaining Good Architecture

Within the framework of research-oriented software development, the typical attitude taken by researchers is to treat software development as a means to an end, where the time to a working prototype (i.e., the "quick and dirty" approach) is valued over elegance in design. At some point, everyone who writes code is faced with the age-old dilemma: do I just hardcode it (i.e., explicitly program the solution for a specific problem), or do I devise a generic solution to the problem? While the former is often easier, you may quickly find yourself thinking, "I have a bad feeling about this." When designing your software, it is important to keep the big picture in mind. A quick and dirty solution may get the job done now, but what kind of capabilities will you need later? Is your solution scalable if more features are added? If you have already carefully thought out the intended scope of the project (Lesson 2), then some of the potential challenges faced by scaling up can be eliminated up front.

In Lesson 4, we discussed the concept of design patterns and why it is important to think about them before you start implementing any code. It is also important to think ahead when you sit down and start to implement code. One principle of good software architecture is to ensure that your design is modular. In other words, try to separate various functions into independent modules that are tied together using the design pattern of your choice. How does this work practically? Before diving headfirst into implementation, take a few minutes to think about how you can modularize your design. Try using a whiteboard to sketch out a few boxes that represent the major pieces of your system. Think about how these modules will fit together and communicate with each other. For example, if your modules can be split into functions or objects, what are the inputs and outputs of each function?

One mark of a good developer is the willingness and patience to rewrite and reorganize code when necessary. It is helpful to go through this process (referred to as "refactoring") after you have completed a working prototype for a given module. Refactoring ultimately helps your code be better organized and more accessible to others. Further, since you are now rewriting code whose structure is a little clearer (now that you are writing it for at least the second time), the refactoring process provides a good opportunity to test (Lesson 9) and document (Lesson 10) your code.

Lesson 9: Thoroughly Test Your Code

Although testing sounds like something you do after writing functional software, many software developers start writing tests from the beginning of development, or, in some cases, before writing any code at all! Testing has enormous benefits: you can rewrite important pieces of your codebase and see immediately whether it still works, verify that collaborators have not broken key functionality with their contributions, check for edge cases, quickly check whether your software runs on a new platform, or check that new versions of dependencies will not have unexpected consequences.

Please cite this article in press as: Yurkovich et al., A Padawan Programmer's Guide to Developing Software Libraries, Cell Systems (2017), https://doi.org/10.1016/j.cels.2017.08.003

Table 2. Common Software Licenses

License	Description	When to choose
MIT License	Least restrictive, allows users to do whatever they want provided they reference you and do not hold you liable.	If you want the simplest option and are not concerned with intellectual property.
Apache License	Similar to the MIT license, provides patent rights.	If you are concerned about patents and intellectual property.
GNU General Public License (GPL)	Requires those who distribute your code or build extensions to make the source available under identical terms.	If you are concerned about how extensions to your software are licensed and managed.
GNU Lesser General Public License (LGPL)	Allows third parties to use and integrate your software into their own (proprietary) software provided that this component is modifiable and redistributed under identical terms.	If you require source-code redistribution and are concerned about patents and intellectual property while still allowing free use.

Adapted from https://choosealicense.com/.

When you start to investigate how best to test your software, you will see references to various kinds of tests, especially unit testing, integration testing, and validation testing (Fucci et al., 2016). Unit tests are a good place to start. For any function or method that you define, write a test to makes sure it works the way you expect. Integration tests work with larger pieces of your codebase, testing whether modules will work together as expected. Validation tests check whether your entire codebase meets specifications. There are also methodologies to help structure writing tests—such as the popular test-driven development (TDD) (Beck, 2003)—that might help you get started.

Lesson 10: Comment Your Code and Write Documentation

One of the primary reasons to make software open source is to enable others in the community to contribute to your project. But have you ever tried to read someone else's code? (It's a trap!) If you have, you know that thoroughly commented code and good documentation is vital. The most important piece of advice we can give: write your documentation as you go. Every time you add a new feature, make sure that you leave a comment—try to make it a habit. It is important to consider technical documentation on specific features and interfaces as well as general documentation for new users. Another good practice is to answer frequently asked questions in the documentation; if someone asks you a question, take a few extra minutes to make sure the answer you give them is also in the documentation. Good documentation may include cross-references to other points in the text, screenshots to explain step-by-step procedures, and code examples that demonstrate how to access various functionality.

Lesson 11: Get User Input

Once you have achieved a development goal and have a working prototype, it is important to get user input. This is an opportunity to test drive your documentation as well as your code. Is the interface intuitive and easy to use? Does your module have the functionality in which users are most interested? Is the documentation you have written helpful? Are users able to follow the comments and documented examples to get the most out of the software's features? The most helpful input will likely come from your own collaborators and colleagues—if the people from whom you seek input are not likely to use your software, then these are not the people you are looking for.

Once you have gotten some useful feedback, you can move back to the prototyping stage if necessary (Lesson 8) to implement some of the suggestions. If users are satisfied with the product, then it is time to move forward. Depending on how far along your project is, you may take one of two routes from here (Figure 1): either identify the next development goal (Lesson 3) and continue the software development process or start thinking about code distribution (Lessons 12–14).

Lesson 12: Understand Licensing Issues

Sharing code is not quite as easy as it sounds. It only takes a second to paste code into a message board, but sharing code does not necessarily give others a legal right to use or adapt it. Therefore, it is important to provide a license with any significant piece of code that you share.

Luckily, it is also easy to add a license. A lot of academic software is free and open-source software (FOSS), meaning that the software is free for anyone to use or modify for any purpose. There are many options for licensing your software—popular options include the MIT, Apache, and General Public Licenses (Table 2)—and it can require some research to find which is right for you (St. Laurent, 2004). Previous articles have discussed licensing options for software (Prlić and Procter, 2012) and data (Goodman et al., 2014). The Open Source Initiative (https://opensource.org) provides many resources on the subject. GitHub also provides an excellent resource (https://choosealicense.com) for getting started with a software license, with descriptions of licenses and when to choose which license.

Lesson 13: Share Your Code

Once you have a license for your code, you are ready to start distributing it to the community at large. Among the best and widely used methods for code distribution are structured content versioning repositories like GitHub and BitBucket. Although something like GitHub is great for sharing code, it is also a great tool to use throughout the development process. The built-in issue tracker enables you (and other users) to keep track of bugs, feature requests, and code contributions. When you share your code, try to include as much information as you can about the goals of the project, the development

Table 3. Tools and Resources Discussed in This Article

Tool/Resource	Description	Link
GitHub and BitBucket	Web-based version control for hosting and distributing code.	https://github.com; https://bitbucket.org
Jupyter	Application to create notebooks with live code, equations, visualizations and explanatory text, with support for over 40 programming languages (e.g., Julia, Python, R).	http://jupyter.org
R/Shiny	A web application framework for the R programming language with support for notebooks and data dashboards.	https://shiny.rstudio.com
Slack	Cloud-based team collaboration and project organization tools and services with extensive messaging capabilities.	https://slack.com
Trello	Collaboration tool for project organization and management, with a simple card-based interface.	https://trello.com
PivotalTracker	Project organization and management tools for team-based collaboration.	https://www.pivotaltracker.com
Semantic Versioning	Guidelines for software versioning to minimize and communicate breaking changes.	http://semver.org
Hello World Tutorial	Resource offered through GitHub that explains version control via examples.	http://guides.github.com/activities/hello-world
Read the Docs	Fully-searchable documentation hosting service that allows for version control.	https://readthedocs.org
Open Source Initiative	Provides many resources for the development and dissemination of open-source software.	https://opensource.org
Choose a License	Resource offered through GitHub that provides information about various licenses.	https://choosealicense.com

status, who the target users are, how others should report bugs and contribute code, and whom they should contact if they have additional questions. If you take the time to regularly respond to inquiries, you will give your library a much better chance of catching on.

Lesson 14: Market Your Tool

Now that you have developed software that is intuitive, extensible, and easy to use, you need to spread the word! Reach out to potential users and ask them to try it out. Bring demos, screenshots, or figures with you to meetings and conferences. Consider publishing your software in an academic journal or preprint server, as this helps raise awareness and gives users the chance to cite your work. Something that is easily overlooked is how you brand your tool; a good name is easy to pronounce, memorable, and easily found by an internet search.

The next step is to get the community involved. Even if you enjoy personally developing your software, it can be extremely valuable to build a community around it. By sharing your work with others, you can achieve much more than you could alone. Not only will you be able to infuse fresh ideas and find new use cases for your tool but you will discover bugs in places you did not even think to test. Furthermore, having multiple developers will increase the visibility of your work, because all contributors will work to make the project visible.

Community software development brings in a new set of challenges. You will need to establish rules for contributing code, a style guide so code is consistent and readable, a process for reviewing new code contributions, and more. There are many tools, resources, and references out there to assist in this process (Table 3). The challenges of leading an open-source software project would be an excellent topic for a future article.

ACKNOWLEDGMENTS

The authors gratefully acknowledge Niko Sonnenschein for reading the manuscript and providing valuable feedback. Possible this manuscript would not have been without Jared Broddrick and Laurence Yang. This research was supported by the US Department of Energy (DE-SC0008701), by the Institutional Strategy of the University of Tübingen (German Research Foundation DFG, ZUK 63), and by the Novo Nordisk Foundation through the Center for Biosustainability at the Technical University of Denmark (NNF10CC1016517).

REFERENCES

Altschul, S., Demchak, B., Durbin, R., Gentleman, R., Krzywinski, M., Li, H., Nekrutenko, A., Robinson, J., Rasband, W., Taylor, J., and Trapnell, C. (2013). The anatomy of successful computational biology software. Nat. Biotechnol. *31*, 894–897.

Altschul, S.F., Madden, T.L., Schäffer, A.A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D.J. (1997). Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res. *25*, 3389–3402.

Beck, K. (2003). Test-driven Development: By Example (Addison-Wesley Professional).

Berman, H.M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T.N., Weissig, H., Shindyalov, I.N., and Bourne, P.E. (2000). The Protein Data Bank. Nucleic Acids Res. *28*, 235–242.

Bordbar, A., Nagarajan, H., Lewis, N.E., Latif, H., Ebrahim, A., Federowicz, S., Schellenberger, J., and Palsson, B.O. (2014). Minimal metabolic pathway structure is consistent with associated biomolecular interactions. Mol. Syst. Biol. *10*, 737.

Bordbar, A., Yurkovich, J.T., Paglia, G., Rolfsson, O., Sigurjónsson, Ó.E., and Palsson, B.O. (2017). Elucidating dynamic metabolic physiology through network integration of quantitative time-course metabolomics. Sci. Rep. *7*, 46249.

Brazma, A., Krestyaninova, M., and Sarkans, U. (2006). Standards for systems biology. Nat. Rev. Genet. *7*, 593–605.

Brown, W.J. (1998). AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis (John Wiley & Sons Incorporated).

Chung, H.-J., Park, C.H., Han, M.R., Lee, S., Ohn, J.H., Kim, J., Kim, J., and Kim, J.H. (2005). ArrayXPath II: mapping and visualizing microarray gene-expression data with biomedical ontologies and integrated biological pathway resources

using Scalable Vector Graphics. Nucleic Acids Res. *33*, W621–W626.

Dräger, A., and Palsson, B.Ø. (2014). Improving collaboration by standardization efforts in systems biology. Front. Bioeng. Biotechnol. *2*, 61.

Droste, P., Nöh, K., and Wiechert, W. (2013). Omix - a visualization tool for metabolic networks with highest usability and customizability in focus. Chemieingenieurtechnik (Weinh.) *85*, 849–862.

Dudley, J.T., and Butte, A.J. (2009). A quick guide for developing effective bioinformatics programming skills. PLoS Comput. Biol. *5*, e1000589.

Ebrahim, A., Lerman, J.A., Palsson, B.O., and Hyduke, D.R. (2013). COBRApy: constraints-based reconstruction and analysis for python. BMC Syst. Biol. *7*, 74.

Federhen, S. (2012). The NCBI Taxonomy database. Nucleic Acids Res. *40*, D136–D143.

Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., and Juristo, N. (2016). A dissection of test-driven development: does it really matter to test-first or to test-last? IEEE Trans. Softw. Eng. *43*, https://doi.org/10.1109/TSE.2016.2616877.

Funahashi, A., Matsuoka, Y., Jouraku, A., Morohashi, M., Kikuchi, N., and Kitano, H. (2008). CellDesigner 3.5: a versatile modeling tool for biochemical networks. Proc. IEEE *96*, 1254–1265.

Geer, L.Y., Marchler-Bauer, A., Geer, R.C., Han, L., He, J., He, S., Liu, C., Shi, W., and Bryant, S.H. (2010). The NCBI BioSystems database. Nucleic Acids Res. *38*, D492–D496.

Goodman, A., Pepe, A., Blocker, A.W., Borgman, C.L., Cranmer, K., Crosas, M., Di Stefano, R., Gil, Y., Groth, P., Hedstrom, M., et al. (2014). Ten simple rules for the care and feeding of scientific data. PLoS Comput. Biol. *10*, e1003542.

Heirendt, L., Thiele, I., and Fleming, R.M.T. (2017). DistributedFBA.jl: high-level, high-performance flux balance analysis in Julia. Bioinformatics. https://doi.org/10.1093/bioinformatics/btw838.

Hunt, A., and Thomas, D. (1999). The Pragmatic Programmer: From Journeyman to Master (Addison-Wesley Professional).

Johnson, R., Gamma, E., Vlissides, J., and Helm, R. (1995). Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley).

Kelder, T., van Iersel, M.P., Hanspers, K., Kutmon, M., Conklin, B.R., Evelo, C.T., and Pico, A.R. (2012). WikiPathways: building research communities on biological pathways. Nucleic Acids Res. *40*, D1301–D1307.

King, Z.A., Dräger, A., Ebrahim, A., Sonnenschein, N., Lewis, N.E., and Palsson, B.O. (2015). Escher: a web application for building, sharing, and embedding data-rich visualizations of biological pathways. PLoS Comput. Biol. *11*, e1004321.

Kono, N., Arakawa, K., Ogawa, R., Kido, N., Oshita, K., Ikegami, K., Tamaki, S., and Tomita, M. (2009). Pathway projector: web-based zoomable pathway browser using KEGG atlas and Google Maps API. PLoS ONE *4*, e7710.

Leprevost, F. da V., Barbosa, V.C., Francisco, E.L., Perez-Riverol, Y., and Carvalho, P.C. (2014). On best practices in the development of bioinformatics software. Front. Genet. *5*, 199.

List, M., Ebert, P., and Albrecht, F. (2017). Ten simple rules for developing usable software in computational biology. PLoS Comput. Biol. *13*, e1005265.

Loman, N., and Watson, M. (2013). So you want to be a computational biologist? Nat. Biotechnol. *31*, 996–998.

Prlić, A., and Procter, J.B. (2012). Ten simple rules for the open development of scientific software. PLoS Comput. Biol. *8*, e1002802.

Schellenberger, J., Que, R., Fleming, R.M.T., Thiele, I., Orth, J.D., Feist, A.M., Zielinski, D.C., Bordbar, A., Lewis, N.E., Rahmanian, S., et al. (2011). Quantitative prediction of cellular metabolism with constraint-based models: the COBRA Toolbox v2.0. Nat. Protoc. *6*, 1290–1307.

Shannon, P., Markiel, A., Ozier, O., Baliga, N.S., Wang, J.T., Ramage, D., Amin, N., Schwikowski, B., and Ideker, T. (2003). Cytoscape: a software environment for integrated models of biomolecular interaction networks. Genome Res. *13*, 2498–2504.

Smoot, M.E., Ono, K., Ruscheinski, J., Wang, P.-L., and Ideker, T. (2011). Cytoscape 2.8: new features for data integration and network visualization. Bioinformatics *27*, 431–432.

Stanford, N.J., Wolstencroft, K., Golebiewski, M., Kania, R., Juty, N., Tomlinson, C., Owen, S., Butcher, S., Hermjakob, H., Le Novère, N., et al. (2015). The evolution of standards and data management practices in systems biology. Mol. Syst. Biol. *11*, 851.

St. Laurent, A.M. (2004). Understanding Open Source and Free Software Licensing: Guide to Navigating Licensing Issues in Existing & New Software (O'Reilly Media, Inc.).

Via, A., Blicher, T., Bongcam-Rudloff, E., Brazas, M.D., Brooksbank, C., Budd, A., De Las Rivas, J., Dreyer, J., Fernandes, P.L., van Gelder, C., et al. (2013). Best practices in bioinformatics training for life scientists. Brief. Bioinform. *14*, 528–537.

Wilson, G., Aruliah, D.A., Brown, C.T., Chue Hong, N.P., Davis, M., Guy, R.T., Haddock, S.H.D., Huff, K.D., Mitchell, I.M., Plumbley, M.D., et al. (2014). Best practices for scientific computing. PLoS Biol. *12*, e1001745.